PART 1

Getting Started With Windows Powershell Basics

CHAPTER 1 WINDOWS POWERSHELL 101 03

CHAPTER 2 FILTERING AND ITERATING YOUR DATA 39

CHAPTER 3 MAKING YOUR POWERSHELL REUSABLE 65

COPYRIGHTED MATERIAL

CHAPTER 1

Windows PowerShell 101

IN THIS CHAPTER, YOU WILL LEARN TO:

PREPARE FOR YOUR JOURNEY 4

Automate SharePoint: A Historical Perspective 5

From STSADM to Windows PowerShell 7

UNDERSTAND THE COMPONENTS 8

The SharePoint 2010 Management Shell 8

Understand Required Permissions 11

Windows PowerShell: An Object-Based Scripting Language . . . 12

Discover Cmdlets and Objects 14

UNDERSTAND VARIABLES AND TYPES 21

Declare and Use Variables 21

Understand Object Types 22

Declare and Use Arrays and Hash Tables 28

WORK WITH OUTPUT 31

Understand the Object Pipeline and PipeBind Objects 31

Format Your Results 35

CHAPTER 1

Microsoft's latest version of its SharePoint platform, *SharePoint 2010*, introduces numerous new capabilities and architectural changes that make what was already a powerful platform even more powerful and, in doing so, much more complex and difficult to manage. As with the 2007 version of SharePoint, you can use the point-and-click administrative tools found in the SharePoint Central Administration website or you can use *STSADM*, SharePoint 2007's command-line tool. But with SharePoint 2010, we now have built-in support for *Windows PowerShell*, an incredibly powerful scripting tool meant to replace the old batch files that many have used for years.

Before you get started managing and manipulating SharePoint using Windows PowerShell, it is useful to understand the relationship between STSADM and PowerShell as well as core PowerShell concepts that every PowerShell "programmer" must know. In this chapter, we will cover four key areas to help you get started:

- Going from STSADM to Windows PowerShell
- Windows PowerShell components
- Variables and types
- Working with output

Prepare for Your Journey

Before you can begin working with SharePoint 2010 and Windows PowerShell, you must first get your environment configured and ready for use. The full setup and configuration of your environment is out of scope for this book; we will be focusing on deploying and managing your SharePoint environment using Windows PowerShell and not the setup and configuration of your operating system and related installation files.

In general, you will want to make sure that you, at a minimum, have the SharePoint 2010 binaries installed, have installed Windows PowerShell 2.0, and are logged on with an account with appropriate rights. (See "Understand Required Permissions" later in this chapter for more details.) It is also assumed that the server is a member of a SharePoint 2010 Farm, except where explicitly noted.

NOTE **Windows PowerShell 2.0 will be automatically installed when you install the SharePoint 2010 prerequisites. Though it is possible to use Windows PowerShell 1.0 with SharePoint 2010, it is highly recommended that you do not.**

Before you dig deep into Windows PowerShell, it is a good idea to take a look at the other scripting options available to you, specifically STSADM and *PSConfig*, the command-line version of the SharePoint Configuration Wizard. This is useful because there are some tasks that can only be accomplished using STSADM (though not many).

Automate SharePoint: A Historical Perspective

When SharePoint 2003 was introduced, Microsoft made available the first version of STSADM. This allowed administrators to automate some common tasks related to the day-to-day management and deployment activities of SharePoint. Unfortunately, the tool was very limited in scope and not extensible. With SharePoint 2007, Microsoft introduced numerous new commands (over 184 in total) and adapted the architecture of the tool so that third-party developers could extend it by adding new commands. This provided a consistent and uniform administrative experience.

Administrators (and developers) would very often create batch files that would call out to these various STSADM commands. Unfortunately, batch files provided limited support for looping and conditional expressions, so it was common for batch files to become very complex and error-prone.

The following is a simplified version of a snippet from a SharePoint 2007 Farm creation script:

```
ECHO %DATE% %TIME%:  Building configuration database
psconfig -cmd configdb -create -server %SERVER_DB% ↩
  -database %DB_CONFIG_NAME% -user %ACCT_SPFARM% ↩
  -password %ACCT_SPFARM_PWD% -admincontentdatabase ↩
  %DB_CENTRALADMINCONTENT_NAME%
if not errorlevel 0 goto errhnd

ECHO %DATE% %TIME%: Installing help content
psconfig -cmd helpcollections -installall
if not errorlevel 0 goto errhnd

ECHO %DATE% %TIME%: Securing resources
psconfig -cmd secureresources
if not errorlevel 0 goto errhnd

ECHO %DATE% %TIME%: Installing services
```

```
psconfig -cmd services -install
if not errorlevel 0 goto errhnd

ECHO %DATE% %TIME%: Installing features
psconfig -cmd installfeatures
if not errorlevel 0 goto errhnd

ECHO %DATE% %TIME%: Creating central admin site
psconfig -cmd adminvs -provision -port ↵
  %CENTRALADMIN_PORT% -windowsauthprovider enablekerberos
if not errorlevel 0 goto errhnd

ECHO %DATE% %TIME%: Adding app content to CA site
psconfig -cmd applicationcontent -install
if not errorlevel 0 goto errhnd

goto end

:errhnd

echo An error occurred - terminating script.

:end
```

In this case, the script is using `PSConfig` to create a SharePoint 2007 Farm. Notice that variables can be used and follow the format *`%NAME%`*. Here's another example demonstrating the use of STSADM to deploy a custom SharePoint Solution Package to the Farm:

```
stsadm -o addsolution -filename ↵
  "Lapointe.SharePoint.STSADM.Commands.wsp"
stsadm -o deploysolution -local -allowgacdeployment -name ↵
  "Lapointe.SharePoint.STSADM.Commands.wsp"
stsadm -o execadmsvcjobs
```

Any of the preceding scripts can be run from a standard command prompt by simply adding the path of the SharePoint Root BIN folder to the *`%PATH%`* environment variable:

```
c:\program files\common files\microsoft shared\web server
extensions\12\bin
```

As you transition to SharePoint 2010, you will be pleased to note that all of the preceding scripts will work just fine (with the exception of the `psconfig -cmd configdb` command, which now requires a passphrase parameter). The main difference is that the STSADM and PSConfig executable files are now in the 14 hive's BIN folder:

```
c:\program files\common files\microsoft shared\web server
extensions\14\bin
```

The combined use of STSADM and PSConfig allowed administrators the ability to automate numerous tasks, including the complete build-out of an entire Farm. When these command-line tools were coupled with third-party extensions, there was very little that an administrator couldn't automate, but there were limitations:

- The out-of-the-box commands, though many, did not cover numerous aspects of the product and did not expose many parameters.
- STSADM was very slow and did not handle bulk operations well.
- Everything passed into a command was a string, so all type safety was removed and objects had to be continually re-created internally.
- Though Windows PowerShell could have been used as the host, batch files were the standard and were very limited.
- Error handling was difficult.

Eventually, some savvy administrators and developers caught on to PowerShell and started transitioning their batch files over to PowerShell scripts. This not only provided new levels of flexibility and power to what could be done with STSADM, it also helped expose the SharePoint API to administrators without the need to create a .NET assembly. Unfortunately, we still suffered the core limitations of STSADM as well as a host of new issues. Administrators and developers needed more. They needed direct and out-of-the-box support for the Windows PowerShell scripting language. They needed to move away from STSADM.

From STSADM to Windows PowerShell

When SharePoint 2010 was introduced to the market, one of the greatest new features of the product was the addition of over 530 PowerShell *cmdlets* (pronounced "command-lets"). Using these cmdlets in conjunction with the SharePoint API, administrators and developers can now automate or manipulate just about every aspect of the SharePoint platform. Windows PowerShell is Microsoft's

next-generation scripting language and offers direct access to .NET objects, complex flow and structure capabilities, and an object pipeline that makes passing objects from one command to another extremely easy. STSADM and PSConfig are still part of the product, and every command that was still relevant to the product has remained (that is, commands pertaining to things such as the Shared Services Provider, which no longer exists, have been removed).

As you transition over to SharePoint 2010, some of the scripts you may have created for SharePoint 2007 will continue to work with little to no modification. That being said, STSADM has been officially deprecated by Microsoft and may not appear in the next version of SharePoint, so it is a good idea to start reducing or eliminating your dependency on STSADM.

The following sections will help you with this transition by covering core concepts such as how to load Windows PowerShell, how to find what is available to you, and of course, how to get stuff done.

Understand the Components

Before you start working with Windows PowerShell, it's helpful to understand the various components involved and how to find them. The first and most prominent piece is the console application that hosts the PowerShell runtime, the second is PowerShell itself, and the third is the PowerShell and SharePoint cmdlets that do the bulk of the work. Once you are familiar with these various components, the rest is just syntax and structure. The following sections will cover each of these three key elements.

The SharePoint 2010 Management Shell

To use Windows PowerShell with SharePoint, you must first make the Windows PowerShell environment aware of the SharePoint cmdlets and assemblies. The easiest way to do this is to open Windows PowerShell using the *SharePoint 2010 Management Shell*, which you can find under the Microsoft SharePoint 2010 Products folder in the Start menu. The SharePoint 2010 Management Shell automatically registers all the SharePoint 2010 PowerShell cmdlets.

To understand how the SharePoint 2010 Management Shell works, it is helpful to look at what is happening when you open it. If you right-click the shortcut and select Properties, you'll see the following for the Target field:

```
C:\...\PowerShell.exe -NoExit  " & ' ↵
C:\...\14\CONFIG\POWERSHELL\Registration\sharepoint.ps1 ' "
```

The SharePoint 2010 Management Shell is just a Windows PowerShell instance that loads the `sharepoint.ps1` script file. If you open this script file, you will see the following:

```
$ver = $host | select version
if ($ver.Version.Major -gt 1) {
  $Host.Runspace.ThreadOptions = "ReuseThread"
}
Add-PsSnapin Microsoft.SharePoint.PowerShell
Set-location $home

# SIG # Begin signature block
# MIIXUAYJKoZIhvcNAQcCoIIXQTCCFz0CAQExCzA...
# SIG # End signature block
```

The line with the call to `Add-PsSnapin` is the most critical, and it is the only one that matters right now. (The implications of the previous lines are detailed in Chapter 2, "Filtering and Iterating Your Data.") *Add-PsSnapin* is a Windows PowerShell cmdlet that is used to register a collection of new cmdlets that are packaged in what is called a *snap-in*. When the SharePoint 2010 binaries are installed, the assemblies that contain the SharePoint PowerShell cmdlets are also installed and registered with the system. However, those assemblies, and the cmdlets defined within them, will not be available to PowerShell until they are added using the `Add-PsSnapin` cmdlet.

The significance of this is that you can easily work with the SharePoint PowerShell cmdlets in editors other than the SharePoint Management Shell (or, specifically, the Windows PowerShell console window). In fact, our general recommendation is to not use the SharePoint Management Shell and instead to use a script editor such as the *Windows PowerShell Integrated Scripting Environment* (*ISE)* or a third-party scripting editor. The Windows PowerShell ISE is available only with PowerShell V2 and can be loaded by going to Start ➢ All Programs ➢ Accessories ➢ Windows PowerShell. By default, the Windows PowerShell ISE is not available on the Server operating systems and must be added as an additional Feature before it will be available.

If you choose to use the Windows PowerShell ISE, you must manually execute the `Add-PsSnapin Microsoft.SharePoint.PowerShell` command. However,

you will now have the benefit of integrated debugging capabilities, syntax highlighting, and of course, a simple editor in which you can construct your routines without having to execute them line by line.

An alternative to manually executing the `Add-PsSnapin` cmdlet would be to update your default user profile. This will enable you to use the SharePoint PowerShell cmdlets in any editor for any user. To update your profile, run the following from any Windows PowerShell console window or the Windows PowerShell ISE to verify and create your profile script:

```
if (!(Test-Path $profile.AllUsersAllHosts)) {
  New-Item -Type file -Path $profile.AllUsersAllHosts -Force
}
```

Now that the profile script exists, you can edit it. From the Windows PowerShell ISE, run the following command:

```
Psedit $profile.AllUsersAllHosts
```

This will load your profile script into the editor. If you are in a standard Windows PowerShell window, type **`ise $profile.AllUsersAllHosts`** instead, and this will launch the Windows PowerShell ISE with the script file open for editing. Now add the following text to the script and save the file:

```
if ($host.UI.RawUI.WindowTitle -ne ↩
"Administrator: SharePoint 2010 Management Shell") {
  $ver = $host | select version
  if ($ver.Version.Major -gt 1) {
    $host.Runspace.ThreadOptions = "ReuseThread"
  }
  if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" ↩
-ErrorAction SilentlyContinue) -eq $null) {
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
  }
}
```

This script does basically the same thing as the `sharepoint.ps1` script, but it adds a further check to make sure that it is not run if the host is the SharePoint 2010 Management Shell. It also does a check to make sure the SharePoint PowerShell snap-in isn't already loaded. It's not important that you understand every piece of this script at this time because we'll be covering the concepts demonstrated in this script throughout this chapter and the next.

TIP For more information on editing your user profile script, type **`help about_profiles`** from any Windows PowerShell window or view the TechNet article "How to Create Profiles in Windows PowerShell ISE" at **`http://technet.microsoft.com/en-us/library/dd819492.aspx`**.

Understand Required Permissions

Before you can use the SharePoint 2010 Management Shell (or specifically, the SharePoint 2010 cmdlets), you must have the appropriate permissions. You must be a member of the `SharePoint_Shell_Access` role in the SQL Server database applicable to the commands that are being run, and you must be a member of the `WSS_Admin_WPG` local security group on each SharePoint server. By default, the `SharePoint_Shell_Access` role exists only in the SharePoint Configuration Database.

Using the *Add-SPShellAdmin* cmdlet, you can easily add users to these roles across your Farm. Typically, the setup account (the account used to build the SharePoint Farm) will have rights to run this cmdlet, but you can enable others to run the cmdlet by ensuring that they are a local administrator on the server and have `dbcreator` and `securityadmin` rights on the database servers. If you are using your setup account, there is nothing you need to do.

To grant a user rights to execute most SharePoint cmdlets that affect the Farm, run the following command, which will add the user to the `SharePoint_Shell_Access` role in the Configuration Database and to the `WSS_Admin_WPG` local security group on each server:

```
PS C:\> Add-SPShellAdmin -UserName "domain\user"
```

To grant a user rights to execute cmdlets that affect a specific database, add the `-Database` parameter to the call:

```
PS C:\> Add-SPShellAdmin -UserName "domain\user" ↵
  -Database "DatabaseName"
```

In some cases, you may also be required to add the user to the local administrators group manually on one or more servers. You may also be required to add the user to the Farm Administrators group, which can be done using either the Central Administration site or Windows PowerShell using the `New-SPUser` cmdlet, as shown below:

```
PS C:\> New-SPUser -UserAlias "domain\user" -Web ↵
  "http://sp2010:12345/" -Group "Farm Administrators"
```

In this case, `http://sp2010:12345/` is the URL to the Central Administration site.

Throughout the course of this book, we will assume that you are executing the scripts as a user with sufficient permissions. Namely, you are a Local Administrator or a Farm Administrator, and you have been granted Shell Administrator rights via the `Add-SPShellAdmin` cmdlet. In some cases, you should also be in the `dbcreator` and `securityadmin` roles within SQL Server. Note that, by default, your SharePoint setup account, often known as the *SPAdmin* account, will have all these permissions. Though we don't recommend using this account for everyday use in your production environment, it is often used when developing or demo-ing product features due to its elevated level of access.

Windows PowerShell: An Object-Based Scripting Language

To begin using the SharePoint 2010 PowerShell cmdlets, open the SharePoint 2010 Management Shell. If User Access Control (UAC) has not been disabled, then you must open the SharePoint 2010 Management Shell by right-clicking the link's shortcut and clicking `Run as administrator`. Failure to run the shell as an elevated user will result in the SharePoint cmdlets failing to load properly. All of the required core SharePoint assemblies and the SharePoint 2010 PowerShell snap-in should be registered and available for use. We'll start our exploration of Windows PowerShell with a simple example that demonstrates using the *Get-SPFarm* cmdlet:

```
PS C:\> $farm = Get-SPFarm
PS C:\> $farm.GetType().FullName
Microsoft.SharePoint.Administration.SPFarm
```

If you've worked with the SharePoint object model before, you should immediately recognize the type name `Microsoft.SharePoint.Administration.SPFarm`. The `Get-SPFarm` cmdlet returns back a first-class .NET object that can be used and manipulated just like any other .NET object.

To explore this further, pass the `$farm` variable just created into a standard Windows PowerShell cmdlet called *Get-Member*:

```
PS C:\> $farm | Get-Member

Name                              MemberType Definition
----                              ---------- ----------
AddBackupObjects                  Method System.Void Add...
Clone                             Method System.Object C...
CurrentUserIsAdm...               Method bool CurrentUse...
```

```
...
TimerService                    Property Microsoft.Share...
TraceSessionGuid                Property System.Guid Tra...
TypeName                        Property System.String T...
UpgradeContext                  Property Microsoft.Share...
UpgradedPersiste...             Property System.Collecti...
Version                         Property System.Int64 Ve...
```

The `Get-Member` cmdlet takes in any object and outputs information about the object's public methods and properties. This means that you have a fully featured, real-time compiler that allows you to manipulate virtually any object in a SharePoint Farm, and you have convenient, easy-to-use cmdlets that provide quick access to those objects!

But what if there's no cmdlet that provides access to what you need? No problem—you can still get to any .NET object indirectly by accessing the object via a container (for example, retrieve an `SPList` object using the `SPWeb` object, which can be retrieved using the `Get-SPWeb` cmdlet), directly by loading the object's assembly and creating a new instance of the object, or by using a static method or property. The following example demonstrates this by retrieving the `SPFarm` object directly rather than via the `Get-SPFarm` cmdlet:

```
PS C:\> [System.Reflection.Assembly]::↩
  LoadWithPartialName("Microsoft.SharePoint")

GAC    Version        Location
---    -------        --------
True   v2.0.50727     C:\Windows\assembly\GAC_MSIL\Mic...

PS C:\> $farm = ↩
  [Microsoft.SharePoint.Administration.SPFarm]::Local
```

This code performs a one-time, in-memory load of the `Microsoft.SharePoint` assembly using the static `LoadWithPartialName` method of the `System.Reflection.Assembly` class. It then calls the `SPFarm`'s static `Local` property to retrieve an instance of the `SPFarm` object. The SharePoint assembly is automatically loaded when the snap-in is registered, but you can load just this one assembly individually rather than with the snap-in, if desired. Typically, you won't do this with SharePoint 2010, but if you're still working with SharePoint 2007, this approach is your only option.

In Windows PowerShell, an object's type is represented by wrapping the object in brackets (for example, `[type name]`). This is most commonly seen when accessing a static method or property of a type, such as the `LoadWithPartialName` method or the `Local` property. However, rather than using a period to access the object member, you use a double colon (`::`). Types are explained in more detail later in this chapter.

To instantiate new objects, you use the *New-Object* cmdlet, as shown in the following example that demonstrates creating a new quota template:

```
$webService = [Microsoft.SharePoint.Administration.↵
  SPWebService]::ContentService
$quota = New-Object Microsoft.SharePoint.Administration.↵
  SPQuotaTemplate
$quota.Name = "Team Site"
$quota.StorageMaximumLevel = 2GB
$quota.StorageWarningLevel = 1.5GB
$webService.QuotaTemplates.Add($quota)
```

This example loads an `SPWebService` object via the static `SPWebService.ContentService` property, which is used to save the quota template that is instantiated on the next line via the `New-Object` cmdlet. If you type `Get-Help New-Object` in your Windows PowerShell console, you'll be presented with a detailed explanation of how the cmdlet works, including how to pass arguments into the object's constructor.

NOTE **Notice in the example that the property values are set using the standard `GB` units (could also have been `MB`). Windows PowerShell recognizes these as standard types and automatically converts them to the number of bytes specified. Try it for yourself. Simply enter `1GB` in your Windows PowerShell console window. You should see an output of `1073741824`.**

The trick to being successful with Windows PowerShell is to learn the object model of the application you want to manipulate as well as any available out-of-the-box cmdlets; the rest is just syntax. The problem you'll find with SharePoint is that the object model is huge and the number of available cmdlets can be staggering, so don't get frustrated as you begin your journey.

Discover Cmdlets and Objects

When you first start working with SharePoint, one of the first challenges that you will face is finding the cmdlets that are needed to accomplish a particular task. This

can be especially difficult when that task requires the use of several cmdlets that must be executed in a specific order or when a specific cmdlet for the task doesn't exist, thus requiring the use of the SharePoint object model.

In most cases, the best way to locate the cmdlets or .NET objects, methods, or properties that you'll need is to use the SharePoint software development kit (SDK), specifically the Index of SharePoint Server 2010 Windows PowerShell cmdlets (`http://technet.microsoft.com/en-us/library/ff678226.aspx`), or your favorite search engine. However, you can also do some basic searching within the Windows PowerShell console. Two cmdlets facilitate this:

- *Get-Command*
- *Get-Member*

You can use the `Get-Command` cmdlet to return any and all cmdlets registered with the Windows PowerShell console. The parameters for the `Get-Command` cmdlet are listed in Table 1.1.

TABLE 1.1 **`Get-Command` parameters**

Parameter	Description
`Name`	Gets information only about the cmdlets or command elements with the specified name. Wildcards are permitted.
`Noun`	Gets cmdlets and functions with names that include the specified noun. Wildcards are permitted.
`Verb`	Gets information about cmdlets and functions with names that include the specified verb. Wildcards are permitted.
`CommandType`	Gets only the specified types of commands. Use `CommandType` or its alias, `Type`. By default, `Get-Command` gets cmdlets and functions. Valid values are `Alias`, `All`, `Application`, `Cmdlet`, `ExternalScript`, `Filter`, `Function`, and `Script`.
`ArgumentList`	Gets information about a cmdlet or function when it is used with the specified parameters ("arguments"), such as a path. The alias for `ArgumentList` is `Args`.
`Module`	Gets the commands that came from the specified modules or snap-ins. Enter the names of modules or snap-ins, or enter snap-in or module objects. You can refer to this parameter by its name, `Module`, or by its alias, `PSSnapin`. The parameter name that you choose has no effect on the command or its output.
`Syntax`	Gets only specified data about the command element.
`TotalCount`	Gets only the specified number of command elements. You can use this parameter to limit the output of a command.

COMMON PARAMETERS

Every Windows PowerShell cmdlet contains some common parameters. To see more information about those parameters, type `help about_commonparameters`. The common parameters are as follows:

`-Verbose`

`-Debug`

`-WarningAction`

`-WarningVariable`

`-ErrorAction`

`-ErrorVariable`

`-OutVariable`

`-OutBuffer`

`-WhatIf`

`-Confirm`

You will see the common variables `-ErrorAction`, `-ErrorVariable`, and `-Confirm` used extensively in later parts of this book.

This first example demonstrates how to return back all the SharePoint 2010 cmdlets:

```
PS C:\> Get-Command -pssnapin Microsoft.SharePoint.PowerShell
```

Running this example isn't extremely helpful because it returns way too much information.

For this next example, assume that you want to create service applications using PowerShell and therefore you must see all the cmdlets that are related to services. The easiest way to do this is to filter by the common string `SPService`, as shown here:

```
PS C:\> Get-Command -noun SPService* | Sort Noun | ft Name

Name
----
Install-SPService
```

```
Set-SPServiceApplication
Unpublish-SPServiceApplication
Remove-SPServiceApplication
Publish-SPServiceApplication
Get-SPServiceApplication
Get-SPServiceApplicationProxy
Remove-SPServiceApplicationProxy
Remove-SPServiceApplicationProxyGroup
New-SPServiceApplicationProxyGroup
Get-SPServiceApplicationProxyGroup
Remove-SPServiceApplicationProxyGroupMember
Add-SPServiceApplicationProxyGroupMember
Get-SPServiceApplicationSecurity
Set-SPServiceApplicationSecurity
Get-SPServiceContext
Set-SPServiceEndpoint
Get-SPServiceEndpoint
Get-SPServiceInstance
Stop-SPServiceInstance
Start-SPServiceInstance
```

This list is much more manageable because you can see some of the core cmdlets that apply to service applications. Of course, there may be additional cmdlets needed for specific services such as the Metadata Service Application. Try another search for anything with the term `metadataservice` in it:

```
PS C:\> Get-Command -noun *metadataservice* | ↵
  Sort Noun | ft Name

Name
----
New-SPMetadataServiceApplication
Set-SPMetadataServiceApplication
Get-SPMetadataServiceApplication
Set-SPMetadataServiceApplicationProxy
Get-SPMetadataServiceApplicationProxy
New-SPMetadataServiceApplicationProxy
```

Once you have identified the cmdlets you need, you can then use the `Get-Help` cmdlet to learn more about how to use the cmdlets:

```
PS C:\> Get-Help Start-SPServiceInstance

NAME
    Start-SPServiceInstance

SYNOPSIS
    Starts the service instance for a service on a specific ser
    ver or the farm.

SYNTAX
    Start-SPServiceInstance [-Identity] <SPServiceInstancePipeB
    ind> [-AssignmentCollection <SPAssignmentCollection>] [-Con
    firm [<SwitchParameter>]] [-WhatIf [<SwitchParameter>]] [<C
    ommonParameters>]

DESCRIPTION
    Use the Start-SPServiceInstance cmdlet to start the service
    instance for a service on a specific server or on the farm.

    For permissions and the most current information about Wind
    ows PowerShell for SharePoint Products, see the online docu
    mentation (http://go.microsoft.com/fwlink/?LinkId=163185).

RELATED LINKS
    Get-SPServiceInstance

REMARKS
    To see the examples, type: "get-help Start-SPServiceInstanc
    e -examples".
    For more information, type: "get-help Start-SPServiceInstan
    ce -detailed".
```

```
For technical information, type: "get-help Start-SPServiceI
nstance -full".
```

Eventually, you'll get to the point where the available cmdlets fail you and you must resort to working with the object model. Again, the SDK is a great resource for you, but oftentimes, just seeing a list of available methods and properties is enough to get you started.

You can use the `Get-Member` cmdlet to return a listing of all the properties and methods that you can work with. The parameters for the `Get-Member` cmdlet are listed in Table 1.2.

TABLE 1.2 **`Get-Member` parameters**

Parameter	Description
`Name`	Provide the name of one or more properties or methods to return back information about that property or method. If you use the `Name` parameter with the `-MemberType`, `-View`, or `-Static` parameters, `Get-Member` gets only the members that satisfy the criteria of all the parameters.
`Force`	Adds the intrinsic members (`PSBase`, `PSAdapted`, `PSObject`, `PSTypeNames`) and the compiler-generated `get_` and `set_` methods to the display. By default, `Get-Member` gets these properties in all views other than `Base` and `Adapted`, but it does not display them.
`InputObject`	Specifies the object whose members are retrieved. When using an array of objects, setting `-InputObject` via the pipeline results in the members of the individual objects of the array being returned, whereas setting `-InputObject` as a parameter results in the members of the array being returned and not its contents.
`MemberType`	Gets only members with the specified member type. The default is `All`.
`Static`	Gets only the static properties and methods of the object.
`View`	Gets only particular types of members (properties and methods).

To use this cmdlet, you must pass an instance of the target object into it:

```
PS C:\> Get-SPSite http://portal | Get-Member
```

You can pass in additional parameters to the `Get-Member` cmdlet either to filter the results down to just properties, for example, or to see the original members of the .NET object without any extension or adaptation. Type extensions and adapters are explained later in this chapter.

CMDLET ALIASES

There are some cmdlets within Windows PowerShell that have built-in abbreviations known as an ***alias***. For example, rather than typing `Get-Command`, you could simply type `gcm`, and similarly for `Get-Member`, you can type `gm`. There are numerous built-in aliases, and you can see them all by executing the ***Get-Alias*** cmdlet as shown. The following results have been trimmed to show the most common aliases:

```
PS C:\> Get-Alias

CommandType     Name            Definition
-----------     ----            ----------
Alias           %               ForEach-Object
Alias           ?               Where-Object
Alias           asnp            Add-PSSnapIn
Alias           cat             Get-Content
Alias           cd              Set-Location
Alias           chdir           Set-Location
Alias           cls             Clear-Host
Alias           copy            Copy-Item
Alias           del             Remove-Item
Alias           diff            Compare-Object
Alias           dir             Get-ChildItem
Alias           echo            Write-Output
Alias           fc              Format-Custom
Alias           fl              Format-List
Alias           foreach         ForEach-Object
Alias           ft              Format-Table
Alias           fw              Format-Wide
Alias           gm              Get-Member
Alias           ise             powershell_ise.exe
Alias           ogv             Out-GridView
Alias           select          Select-Object
Alias           set             Set-Variable
Alias           sleep           Start-Sleep
Alias           sort            Sort-Object
Alias           start           Start-Process
Alias           tee             Tee-Object
Alias           where           Where-Object
Alias           write           Write-Output
```

Several of these aliases will be used throughout the course of the book.

You can also create new aliases using the `Set-Alias` cmdlet. Type **`help about_aliases`** for more information about using and creating aliases.

Understand Variables and Types

Now that you know the core components involved, you can begin to use Windows PowerShell to perform some tasks. The following sections will review the core Windows PowerShell elements that every SharePoint administrator must know. These include declaring and using variables and working with various data types, including the core value types as well as arrays and hash tables.

Declare and Use Variables

In all programming and scripting languages, storage locations are needed to hold data so that it can be accessed at a later time. These locations are referred to as variables.

Windows PowerShell variables are designated with a dollar sign. For example, declaring a variable that stores the results of the `Get-SPFarm` cmdlet would look like this:

```
PS C:\> $farm = Get-SPFarm
```

The actual variable is `farm`, the `$` in front of it simply lets you know that it is a variable. The variable `$farm` stores an instance of the `Microsoft.SharePoint.Administration.SPFarm` class. This is great because you don't have to think about what type a cmdlet returns. (This is due to the adaptive type system, which is explained later in this chapter.)

TIP **Windows PowerShell supports the ability to require that all variables be explicitly declared in a manner similar to the one used for the "Option Explicit" feature in Visual Basic. An exception would be thrown if a variable is referenced before a value is assigned. To enable this feature, type `set-psdebug -strict`, and to disable it, type `set-psdebug -off`.**

The name of variables can contain letters, numbers, spaces, or special characters. There are several accepted naming conventions that you may adopt. While variables may contain spaces, the use of spaces would require the variable name to be enclosed in curly braces:

```
PS C:\> ${SharePoint Site Name} = "Home"
PS C:\> ${Content Database} = "WSS_Content"
```

While these names will work in Windows PowerShell, it is more common to see variables expressed using camel case. *Camel case* is a naming convention that has the first word lowercased and the first letter of each additional word capitalized. The preceding example would look like the following:

```
PS C:\> $sharePointSiteName = "Home"
PS C:\> $contentDatabase = "WSS_Content"
```

COMBINING STATEMENTS

You can combine Windows PowerShell statements on the same line by using the semicolon. If you are coming from a development language similar to C# or Java, then this concept will be easy to grasp. We will now combine the previous example into one line.

```
PS C:\> $sharePointSiteName = "Home"; $contentDatabase ↩
 = "WSS_Content"
```

You can see the value of any variable by simply entering the variable name at the console or by using the `Write-Output` cmdlet:

```
PS C:\> Write-Output $farm

Name                    Status
----                    ------
SharePoint_ConfigDB     Online
```

TIP To see the list of variables that have been declared, use the **`Get-Variable`** cmdlet. To see more information about built-in (or automatic) variables, enter **`help about_automatic_variables`**.

Understand Object Types

As previously stated, when you declare a variable or get data from a cmdlet, everything you are creating, or retrieving, is a fully qualified .NET object… and by everything, we mean everything! Consider the following example:

```
PS C:\> "Tessa".GetType()

IsPublic IsSerial Name    BaseType
-------- -------- ----    --------
True     True     String  System.Object
```

In this example, we call the `GetType()` method of the string value `"Tessa"`. From this, you can see that `"Tessa"` is a fully qualified `System.String` object.

ESCAPE CHARACTER

Strings can be defined using either double quotes or single quotes. This is a convenience feature that makes it easier to embed double or single quotes in a string without having to escape the quote using the *escape character* (`):

```
PS C:\> "Tessa says: 'Learn PowerShell!'"
Tessa says: 'Learn PowerShell!'
PS C:\> 'Tessa says: "Learn PowerShell!"'
Tessa says: "Learn PowerShell!"
PS C:\> "Tessa says: `"Learn PowerShell!`""
Tessa says: "Learn PowerShell!"
```

The escape character can also be used to break a command across multiple lines by escaping the newline character:

```
PS C:\> $site = Get-SPSite `
>> "http://portal"
>>
PS C:\>
```

Here are a few more examples demonstrating some common built-in types:

```
PS C:\> $true.GetType().FullName
System.Boolean
PS C:\> (23).GetType().FullName
System.Int32
PS C:\> (23.0).GetType().FullName
System.Double
PS C:\> (2,3).GetType().FullName
System.Object[]
```

In the first example, you can see that, to use a Boolean value of True, you use the built-in variable called `$true` (for False, use `$false`). The second and third examples demonstrate how to check the type of an integer or double value by wrapping the value in parentheses. Parentheses are used a lot in PowerShell to control the execution order.

The last example is perhaps the most interesting. It creates an object array, which is explained in more detail in the next section, but the interesting part is that

Windows PowerShell was able to convert the comma-separated list of items to an array dynamically. This is part of Windows PowerShell's *adaptive type system* that facilitates the automatic conversion of one type to another.

The Adaptive Type System

To examine how the adaptive type system works, we start by looking at an earlier example:

```
PS C:\> $farm = Get-SPFarm
PS C:\> $farm.GetType().FullName
Microsoft.SharePoint.Administration.SPFarm
```

In this example, the `Get-SPFarm` cmdlet is returning back an `SPFarm` object, but we didn't have to declare the variable type explicitly as is necessary with most programming languages, such as C#. Windows PowerShell was able to set the variable dynamically to the correct type. We could have explicitly set the type by using what is known as a type literal. Type literals are just the type name surrounded by square brackets: `[Microsoft.SharePoint.Administration.SPFarm]`. The following sets the variable again, but this time, it explicitly sets the variable type. If the `Get-SPFarm` cmdlet should return a different type, then an exception would be thrown:

```
PS C:\> [Microsoft.SharePoint.Administration.↵
  SPFarm]$farm = Get-SPFarm
```

The interesting thing about the adaptive type system is that, as its name suggests, it is able to adapt a value by converting it from one type to another dynamically:

```
PS C:\> [xml]$config = @"
>> <Farm FarmAccount="sharepoint\spfarm"
>>         ConfigDB="SharePoint_ConfigDB"
>>         AdminContentDB="SharePoint_Content_Admin"
>>         DatabaseServer="spsql1"
>>         Passphrase="Pa`$`$w0rd">
>>       <CentralAdmin Port="1234" AuthProvider="NTLM">
>>           <Servers>
>>               <Server Name="spsvr1" />
>>           </Servers>
```

```
>>       </CentralAdmin>
>> </Farm>
>> "@
>> $config.GetType().FullName
>>
System.Xml.XmlDocument
PS C:\>
```

This interesting example demonstrates two concepts. The first is what is known as a *here string*. Here strings are special markers that indicate that everything between them is a complete string (the opening marker, `@"`, and the closing marker, `"@`, must appear on their own lines as shown in the example). This is very handy when trying to construct variables with complex string representations.

The second is that Windows PowerShell was able to convert this string to a `System.Xml.XmlDocument` object automatically by simply prefacing the variable name with the type alias `[xml]`. But the power doesn't end there. Windows PowerShell includes the concept of *object adapters* that essentially take certain "foreign" objects and wrap them in a special .NET type known as the *PSObject*. The `PSObject` then exposes custom methods and properties that make working with the object more user-friendly and "object oriented." There are several object adapters that come with Windows PowerShell and provide specific implementations of the `PSObject`.

To put this in context, Windows PowerShell provides an object adapter for working with `System.Xml.XmlDocument` objects, which means that you can access the `Farm` node of the XML (the root element) by simply accessing the nodes as though they were properties, as shown here:

```
PS C:\> $config.Farm

FarmAccount    : sharepoint\spfarm
ConfigDB       : SharePoint_ConfigDB
AdminContentDB : SharePoint_Content_Admin
DatabaseServer : spsql1
Passphrase     : Pa$$w0rd
CentralAdmin   : CentralAdmin
```

Look at this a little closer, and use the `Get-Member` cmdlet to see the details of the `Farm` property:

```
PS C:\> $config.Farm | gm

   TypeName: System.Xml.XmlElement

Name                 MemberType           Definition
----                 ----------           ----------
ToString             CodeMethod           static stri...
AppendChild          Method               System.Xml....
Clone                Method               System.Xml....
CloneNode            Method               System.Xml....
CreateNavigator      Method               System.Xml....
Equals               Method               bool Equals...
GetAttribute         Method               string GetA...
GetAttributeNode     Method               System.Xml....
GetElementsByTagName Method               System.Xml....
GetEnumerator        Method               System.Coll...
GetHashCode          Method               int GetHash...
GetNamespaceOfPrefix Method               string GetN...
GetPrefixOfNamespace Method               string GetP...
GetType              Method               type GetType()
HasAttribute         Method               bool HasAtt...
InsertAfter          Method               System.Xml....
InsertBefore         Method               System.Xml....
Normalize            Method               System.Void...
PrependChild         Method               System.Xml....
RemoveAll            Method               System.Void...
RemoveAllAttributes  Method               System.Void...
RemoveAttribute      Method               System.Void...
RemoveAttributeAt    Method               System.Xml....
RemoveAttributeNode  Method               System.Xml....
RemoveChild          Method               System.Xml....
ReplaceChild         Method               System.Xml....
SelectNodes          Method               System.Xml....
SelectSingleNode     Method               System.Xml....
SetAttribute         Method               System.Void...
SetAttributeNode     Method               System.Xml....
Supports             Method               bool Suppor...
```

```
WriteContentTo        Method                 System.Void...
WriteTo               Method                 System.Void...
Item                  ParameterizedProperty  System.Xml....
AdminContentDB        Property               System.Stri...
CentralAdmin          Property               System.Xml....
ConfigDB              Property               System.Stri...
DatabaseServer        Property               System.Stri...
FarmAccount           Property               System.Stri...
Passphrase            Property               System.Stri...
```

Notice how the attributes that are defined within the `Farm` XML element show as properties of the object and that the `TypeName` property is `System.Xml.XmlElement`. Do you notice anything missing? How about the `OuterXml` property or the `Name` or `OwnerDocument` properties, all of which are public members of the `System.Xml.XmlElement` class?

This is due to the dynamically created `PSObject` object that wraps the `System.Xml.XmlElement` object and adds all of the child attributes and elements of the XML document as properties, hiding the built-in properties. To be clear, the built-in properties are still there; they're just not showing up when you query for them using the default options of the `Get-Member` cmdlet. To see all of the available properties and methods, use the `-View` parameter when calling the `Get-Member` cmdlet and specify `All`: `$config.Farm | gm -View All`.

There will be times when there is a conflict between the adapted properties and the built-in properties. Consider if the `Farm` element had an attribute named `Name`. If you typed `$config.Farm.Name`, would you get the XML attribute or the built-in property corresponding to the element name? In this scenario, you'd get the XML attribute value, but what if you wanted the built-in value? To get to the built-in value, you would access it via a special property named *PSBase*: `$config.Farm.PSBase.Name`.

Windows PowerShell also has what are known as *type extensions*. Type extensions are modifications to a type that expose some additional functionality and thus make the type work in a more convenient way. Take a look at the members of the `Get-SPProcessAccount` cmdlet to see this in action:

```
PS C:\> Get-SPProcessAccount | gm

   TypeName: Microsoft.SharePoint.Administration.↵
SPProcessAccount
```

```
Name                  MemberType     Definition
----                  ----------     ----------
Equals                Method         bool Equals(System...
GetHashCode           Method         int GetHashCode()
GetType               Method         type GetType()
LookupName            Method         string LookupName()
ToString              Method         string ToString()
ValidateLogonAccount  Method         System.Void Valida...
SecurityIdentifier    Property       System.Security.Pr...
Name                  ScriptProperty System.Object Name...
```

Notice that the `Name` property is of type `ScriptProperty`. If you look up the `SPProcessAccount` type in the SDK, you should note that it does not include a `Name` property. This property is added dynamically by Windows PowerShell when it loads a special file called a type formatter file. These files, and their creation and use, are outside the scope of this book. However, the key thing to take away is that some objects will have "enhancements" to them that go beyond what is provided by the core type.

Declare and Use Arrays and Hash Tables

Oftentimes, when working with Windows PowerShell, there will be a need to create a collection of data that can be used for input to various cmdlets. Windows PowerShell makes it extremely easy to create these collections in the form of either an array or a hash table. An *array* is essentially just a list of items that can be indexed using their position in the list, where the first item is at index zero. *Hash tables* are a collection of name-value pairs that can be indexed using the name of the item.

To create an array in Windows PowerShell, simply wrap the comma-separated items in parentheses preceded by an `@` symbol:

```
PS C:\> $siteUrls = @("http://portal", "http://teams")
PS C:\> $siteUrls
http://portal
http://teams
PS C:\>
```

To access an item within the array, add the index of the item surrounded by square brackets to the end of the variable name:

```
PS C:\> $siteUrls[1]
http://teams
PS C:\>
```

This collection can then be passed into most cmdlets, such as the `Get-SPSite` cmdlet:

```
PS C:\> $siteUrls | Get-SPSite

Url
---
http://portal
http://teams

PS C:\>
```

In this particular example, the Windows PowerShell runtime is looping through each item in the array and calling the `Get-SPSite` cmdlet for each item. The objects written to the pipeline are then grouped together to form a new collection (an array of type `object[]`) that can then be passed into other cmdlets.

Multiple arrays can easily be combined to form a new array:

```
PS C:\> $siteUrls1 = @("http://portal", "http://teams")
PS C:\> $siteUrls2 = @("http://mysites", "http://projects")
PS C:\> $siteUrls1 += $siteUrls2
PS C:\> $siteUrls1
http://portal
http://teams
http://mysites
http://projects
PS C:\>
```

DECLARING EMPTY ARRAYS

Sometimes, you'll need to create an empty array to which you can dynamically add elements for later processing. To declare an empty array, omit the values within the parentheses. You can then add items using the + operator (or the shorthand += operator):

```
PS C:\> $siteUrls = @()
PS C:\> $siteUrls = $siteUrls + "http://portal"
PS C:\> $siteUrls += "http://teams"
PS C:\> $siteUrls
http://portal
http://teams
PS C:\>
```

Hash tables, also known as dictionaries, are declared by wrapping the list of semicolon-separated items in curly braces and defining a "key" for each item by setting the key equal to the value:

```
PS C:\> $sites = @{"Portal" = "http://portal"; ↩
  "Team Sites" = "http://teams"}
PS C:\> $sites

Name                           Value
----                           -----
Portal                         http://portal
Team Sites                     http://teams

PS C:\>
```

The quotation marks around the key portion are only necessary if the name has a space in it. However, it's generally a good practice always to wrap string names in quotes (note that either the key or the value can be any object type and do not have to be strings).

Like arrays, hash tables can easily be combined:

```
PS C:\> $sites1 = @{"Portal" = "http://portal"; ↩
  "Team Sites" = "http://teams"}
PS C:\> $sites2 = @{"My Sites" = "http://mysites"; ↩
  "Project Sites" = "http://projects"}
PS C:\> $sites1 += $sites2
PS C:\> $sites1

Name                           Value
----                           -----
Project Sites                  http://projects
Portal                         http://portal
My Sites                       http://mysites
Team Sites                     http://teams

PS C:\>
```

Individual items in the hash table can then be accessed either using dot notation (thanks to the dictionary adapter) or by wrapping the key name in square brackets and appending to the variable:

```
PS C:\> $sites1.Portal
http://portal
PS C:\> $sites1."Team Sites"
http://teams
PS C:\> $sites1["My Sites"]
http://mysites
PS C:\>
```

Work with Output

After you've declared and set your variable or executed a specific cmdlet, you're likely to need to pass that information into another cmdlet or otherwise display the results to the console in some formatted fashion. In the following sections, we will review passing data between cmdlets and how to format the results in the console.

Understand the Object Pipeline and PipeBind Objects

Many scripting languages provide a mechanism to "pipe" output from one command to another. In the command prompt console, you can simply use the pipe operator (|) to pass information between commands as shown in this example that demonstrates how to find STSADM commands with the text *user* in them:

```
C:\>stsadm -help | find "user"
           add-ecsuserdefinedfunction
           addexemptuseragent
           adduser
           allowuserformwebserviceproxy
           deleteuser
           enumexemptuseragents
           enumusers
           getsiteuseraccountdirectorypath
           migrateuser
           remove-ecsuserdefinedfunction
           removeexemptuseragent
           setsiteuseraccountdirectorypath
           userrole
stsadm.exe -o adduser
           -userlogin DOMAIN\name
```

```
            -useremail someone@example.com
            -username "Your Name"
stsadm.exe -o enumusers -url http://server/site/web
```

The concept of passing data from one command to another is known as *piping*. The Windows PowerShell *pipeline* facilitates this movement of data between commands.

The Object Pipeline

Windows PowerShell not only includes this capability, it extends it greatly in that cmdlets are more aware of the pipeline and can make intelligent decisions about how to use the passed-in data based on the data type. Think of the pipeline as an assembly line where each stage performs some sort of operation and then passes the results of that operation to the next stage.

Consider the following example that uses the `Get-SPWeb` cmdlet and then "pipes" the resultant `SPWeb` objects to the `Where-Object` cmdlet to filter the results to just those that use the new `v4.master` master page:

```
PS C:\> Get-SPWeb http://portal/* | where {$_.MasterUrl↵
  -like "*/v4.master"}

Url
---
http://portal
http://portal/SiteDirectory
http://portal/SearchCenter
http://portal/News
http://portal/Docs
```

This code passes in a wildcard-based URL to the `Get-SPWeb` cmdlet to return back all `SPWeb` objects under the `http://portal/` web application, and then it passes those objects to the `Where-Object` cmdlet. (Notice how this example uses the `where` alias rather than the full name of the cmdlet.) The variable `$_` represents the current object within the script block. (Filtering and iterating are discussed in more detail in Chapter 2, "Filtering and Iterating Your Data.")

The easiest way to determine what data can be piped into a cmdlet is to get help for the cmdlet and look at the description of the various parameters. Execute `help Get-SPWeb -full` to display the following parameter descriptions (among others):

```
PARAMETERS
    -Identity <SPWebPipeBind>
        Specifies the name or full or partial URL of the subsit
```

```
    e. If you use a relative path, you must specify the
    Site parameter.

    A valid URL in the form http://server_name or a relativ
    e path in the form of /SubSites/MySubSite.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       False
    Accept wildcard characters?  false

-AssignmentCollection <SPAssignmentCollection>
    Manages objects for the purpose of proper disposal. Use
     of objects, such as SPWeb or SPSite, can use large amo
    unts of memory and use of these objects in Windows Powe
    rShell scripts requires proper memory management. Using
     the SPAssignment object, you can assign objects to a v
    ariable and dispose of the objects after they are neede
    d to free up memory. When SPWeb, SPSite, or SPSiteAdmin
    istration objects are used, the objects are automatical
    ly disposed of if an assignment collection or the Globa
    l parameter is not used.

    When the Global parameter is used, all objects are cont
    ained in the global store. If objects are not immediate
    ly used, or disposed of by using the Stop-SPAssignment
    command, an out-of-memory scenario can occur.

    Required?                    false
    Position?                    Named
    Default value
    Accept pipeline input?       True
    Accept wildcard characters?  false

-Site <SPSitePipeBind>
    Specifies the URL or GUID of the site collection from w
    hich to list subsites.

    The type must be a valid URL, in the form  of http://se
```

```
rver_name; a GUID, in the form 1234-5678-9807, or an
SPSite object.

Required?                    false
Position?                    Named
Default value
Accept pipeline input?       True
Accept wildcard characters?  false
```

From this snippet, you can see that the `Get-SPWeb` cmdlet can take an `SPAssignmentCollection` object (which is discussed in Chapter 3) via the pipeline, but you cannot pass the `Identity` parameter via the pipeline. Also notice that you can pass in an `SPSitePipeBind` object via the pipeline. So what is a `PipeBind` object?

PipeBind Objects

As you look through the various SharePoint cmdlets, you will notice that most of them have one or more parameters that take a special object type called a `PipeBind` object. Examine again the first parameter from the help text of the `Get-SPWeb` cmdlet shown earlier. Following is the relevant piece:

```
-Identity <SPWebPipeBind>
    Specifies the name or full or partial URL of the subsite.
    If you use a relative path, you must specify the Site
    parameter.

    A valid URL in the form http://server_name or a relative
    path in the form of /SubSites/MySubSite.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       False
    Accept wildcard characters?  false
```

Within SharePoint, many objects (such as the `SPWeb` and `SPSite` objects) can be represented differently. An `SPSite` object, for example, can be represented using either a URL, a GUID (unique identifier), or an actual `SPSite` object. `PipeBind` objects allow you to use either of these representations for the parameter value.

WARNING **One thing to note is that the name is a bit misleading. Not all `PipeBind` parameters can accept input from the pipeline.**

The following example demonstrates how to use the `SPSitePipeBind` object with the `Get-SPWeb` cmdlet to return all the `SPWeb` objects within the specified site collection:

```
$site = Get-SPSite http://portal
$webs = $site | Get-SPWeb
$webs = $site.ID | Get-SPWeb
$webs = "http://portal" | Get-SPWeb
```

As you can see, you first get an `SPSite` object and then use that object to pipe into the `Get-SPWeb` cmdlet using the three possible representations. Each approach returns the same information. This technique allows the cmdlets to remain relatively simple and avoid having to provide different parameter sets for each possible combination, while at the same time, it allows the user to use the information available without having to transform it to something else.

Format Your Results

To display the contents of a variable to the console, enter the variable alone on a single line and the data will be written to the console using a default view. You can also use the `Write-Output` cmdlet to achieve the same thing:

```
PS C:\> Write-Output $farm

Name                    Status
----                    ------
SharePoint_ConfigDB     Online
```

To format the results in a different way, Windows PowerShell provides four cmdlets that can take an object or collection of objects as input and format them in a particular way:

```
PS C:\> Get-Command -verb format

CommandType     Name                Definition
-----------     ----                ----------
Cmdlet          Format-Custom       Format-Custom [[-...
Cmdlet          Format-List         Format-List [[-Pr...
Cmdlet          Format-Table        Format-Table [[-P...
Cmdlet          Format-Wide         Format-Wide [[-Pr...
```

These cmdlets offer a variety of techniques for outputting data, including the ability to specify which properties of the object you'd like to have displayed and the use of wildcards, regular expressions, and script blocks:

```
PS C:\> $site = Get-SPSite http://portal
PS C:\> $site | Format-List Url, Host*, ↵
  {[int]($_.Usage.Storage/1MB)}

Url                          : http://portal
HostHeaderIsSiteName         : False
HostName                     : portal
[int]($_.Usage.Storage/1MB) : 5
```

This example gets an `SPSite` object and uses the `Format-List` cmdlet (which can also be abbreviated using the `fl` alias) to return back the `Url` property using the exact name of the property, all the properties that start with `Host` using the wildcard character `*`, and finally, a *script block* to return the amount of space consumed by the site collection in megabytes.

Script blocks provide a convenient mechanism to add simple or complex logic to the output mechanism. Any amount of code can be included in the script block, which will be executed for each item passed into the cmdlet. You can add a label to the script block column by changing the script block to the following:

```
@{Expression={[int]($_.Usage.Storage/1MB)}; Label="Size"}
```

If you want to see all the properties, pipe the object to the desired formatter without specifying any properties (with the exception of `Format-Wide`, which is meant to show a single property). Now run the same command, but use the `Format-Table` cmdlet (the default view for most object types) and then the `Format-Custom` cmdlet:

```
PS C:\> $site | Format-Table Url, Host*, {[int]($_.Usage.S
torage/1MB)}

Url            HostHeaderIsSi HostName   [int]($_.Usage
               teName                    .Storage/1MB)
---            -------------- --------   -------------
http://portal  False portal              5


PS C:\> $site | Format-Custom Url, Host*, {[int]($_.Usage.
Storage/1MB)}
```

```
class SPSite
{
  Url = http://portal
  HostHeaderIsSiteName = False
  HostName = portal
  [int]($_.Usage.Storage/1MB) = 5
}
```

The `Format-Custom` cmdlet provides an output similar to that of a class declaration for an object-oriented language. Its use is very limited, and most people favor the `Format-List` cmdlet over it.

If you try the same command using `Format-Wide`, you'll receive an exception because the `Format-Wide` cmdlet can take only one property at a time. To demonstrate the `Format-Wide` cmdlet best, consider the following example, which gets the root web located at `http://portal` and pipes all the lists to the `Format-Wide` cmdlet displaying the title of each list:

```
PS C:\> (Get-SPWeb http://portal).Lists | fw Title

Cache Profiles               Content and Structure Rep...
Content type publishing e... Converted Forms
Customized Reports           Documents
Form Templates               Images
List Template Gallery        Long Running Operation St...
Master Page Gallery          Notification List
Pages                        Quick Deploy Items
Relationships List           Reporting Metadata
Reporting Templates          Reusable Content
Site Collection Documents    Site Collection Images
Solution Gallery             Style Library
Suggested Content Browser... TaxonomyHiddenList
Theme Gallery                User Information List
Variation Labels             Web Part Gallery
wfpub                        Workflow Tasks
```

In this example, the alias `fw` was used rather than the full name of the cmdlet. `Format-Wide` will attempt to squeeze as much information as it can into the horizontal display but, in doing so, will omit considerable amounts of information, and it is often not used as a result.